Using Reinforcement Learning to play Connect Four

Raphael Michel Lucas-Raphael Müller Florian Störtz

Abstract

We report on a successful proof-of-principle application of Reinforcement Learning to the game Connect Four, employing a neural network as policy storage. Training of this is achieved by playing repeatedly against a random opponent, yielding a reward depending on the outcome of the games. We achieve winning rates of over 90% *against random players and over* 70% *in self-play but observe weaknesses of our algorithm when subsequently competing against a human player. Building on this, we have implemented a performant image recognition procedure of a Connect Four board using Haar wavelets, thereby paving the way for a smartphone application which assists human players in the game.*

1. Introduction

In this project, we apply methods from Reinforcement Learning in order to play the well-known game *Connect Four*, in which two players take turns in inserting coins from the top into the columns of a 7×6 grid board. The first player who is able to establish four of their coins consecutively within a column, row, or diagonal, wins the game.

It has been shown by Edelkamp and Kissmann that there are exactly $4\,531\,985\,219\,092$ legal positions in this game [1]. In the 1980s, two authors independently found a strategy for perfect playing of the game [2], [3] that is rather complicated to perform. The game has since also been solved by brute force and and strong solvers have been created using minimax or negamax methods.

Our interest in this project is whether Reinforcement Learning methods will allow us to solve the game with less effort – either less computational effort or less effort in algorithm design and implementation.

Additionally, we implement a traditional vision algorithm with the aim to recognize the current game state of a Connect Four board from a photo of the board. This could be seen as a proof of concept for creating a user-friendly smartphone application that assists in playing the game.

2. Methods

2.1. Vision

We provide a framework which features extraction of game information (i.e. board configuration) by processing image data. The board undergoes a variety of image processing steps and is then identified and checked. In case the former process failed, the user is asked to retake the photograph.

2.1.1 Object Detection using Haar-Like Features

The very first step is detection of the connect4 board by using a Haar-Like feature incorporating Haar wavelet functions. This processing step is done within the openCV framework [4]. After the feature has been trained, the framework features the output of a *Haar cascade xml-file* for fast object detection.

Training has been performed on a set of 25 *distinct* pictures which were taken by hand (20) and from the internet (5). Training of a Haar cascade requires many thousands of positive and negative samples[5]. However, if this many samples are not at hand (even downloading all results of google image search would be insufficient), openCV features the generation of a great amount of sample data which is deduced from a set of present images. The available image data will then serve as negative samples after some manipulation, e.g. projective distortion (warping) and rotation can be applied. After several hours of runtime, *cascade.xml* is provided and can be used for image detection easily.

Figure 1. Object detection using a Haar Algorithm.

2.1.2 Filters

Subsequently, the yellow, red and blue channels as well as a predefined background colour channel are extracted. At the time of writing this work, these colours are being compared to hardcoded values, which should be calibrated in situ using images taken from the respective setup. The binary channels (i.e. red, yellow, blue, background) are obtained by a thresholded RGB color comparison. Further

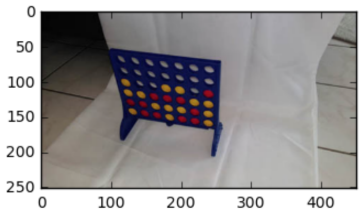

Figure 2. Original photograph before image processing.

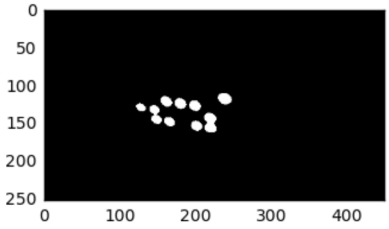

Figure 3. Extraction of a binary red channel.

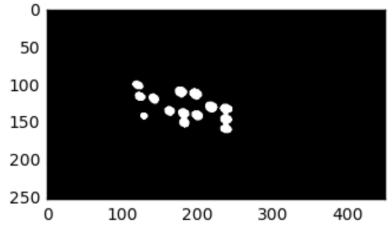

Figure 4. Extraction of a binary yellow channel.

optimisation steps ensue, e.g. to make sure that the background channel only carries the information about empty game slots. This is done by rejecting all background pixels which are not in all four coordinate directions surrounded by board (i.e. blue) pixels.

2.2. Transformation

In the present case, the corner coordinates of the connect4 board are needed for the perspective transformation of the board to a standardized one, from which the information about the allocation of the game slots to the different colours can be extracted. We take the top-right and bottom-left corner pixels to be those active board pixels which have the highest resp. lowest coordinate sum. The other two are obtained in the same way from the board which is mirrored on the y-axis. Using these four points, the colour channels can be transformed back into a standardised rectangular shape, from which the colour allocation is easily extracted.

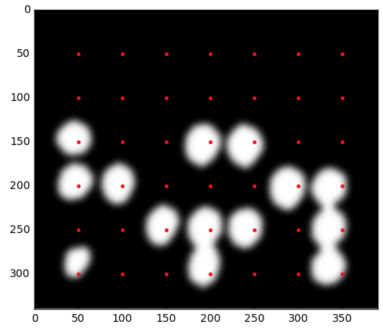

Figure 5. Gridpoints of a regular connect4 board.

2.3. State representation

In theory, we need at least $\lfloor \log_2 3^{6 \cdot 7} \rfloor = 67$ bit to store the entire game state, so we could just use a single 128-bit integer to store it. However, we chose a slightly longer representation by storing the state s in an array of 7 16-bit integers s_i, one for each column (14 bytes or 112 bit in total). In each of these integers s_i we assigned two bits for each row of the board, starting at the lowest two bits with the lowest row. For every cell, we set the assigned bits to 01 to represent a coin of player 1, 10 to represent a coin of player 2 and 00 to represent an empty cell.

We can now efficiently check if e.g. a column is full by just checking if the bits of the highest row are set or if the integer value of the column exceeds $0b01000000000$. Modifying the board and checking for winning positions now just requires simple bitwise operations on these integers.

2.4. Learning

Our approach to learning the game follows standard reinforcement learning methods. Our training algorithm performs N_i iterations and simulates N_g games in every iteration. In each of the simulated games, the moves are determined by a neural network so as to account for the vast amount of possible board states. The current board state is taken as the input to the neural network and one of the allowed next moves is determined randomly, weighted by the output of the neural network. We have only added one single bit of game knowledge to this algorithm: If one of the allowed next moves is a winning move, that move will always be performed. After the move has ended, a total reward of the end position will be calculated. We used a reward of 100 for winning, -100 for losing and -10 for a draw.

This reward will then be used by the employed algorithms for automatic differentiation and optimization[6], implemented by the PyTorch library [7].

The neural network is constructed of a layer of 42 input neurons, one for each cell of the board, then two layers of 64 neurons each and finally a layer of 7 output neurons, one for each column of the board that the next move could put a coin into.

During the simulations, we experimented with three different types of training opponents: random play, random play with winning move recognition, and self-play. In the first case, the opponent just randomly chooses one of the possible moves whilst in the second case, a random move is chosen except in situations where there is a move that leads to a win of the opponent. In the third case, the same algorithm and neural network that we train and use for the first player is used for playing both sides (but only the actions of the first player are used for learning). To avoid the self-play method to just play the same game over and over again, we add some random noise to the decisions of the opponent.

It can be shown that with perfect play on both sides, the first player to insert a coin will always win if and only if they insert their coin in one of the columns adjacent to the middle. If the first coin is inserted in the outer columns of the game however, the second player can win with perfect play.

The consequence of this is that if our learning agent learns the perfect strategy, we would still only see a winning rate of 50% if we randomly let the agent and its opponent begin. For simplicity and more understandable results, we therefore assume that our agent always begins the game.

3. Results

3.1. Vision

The complete image processing time takes < 0.3s (i7-4790K CPU @ 4.00GHz) per image.

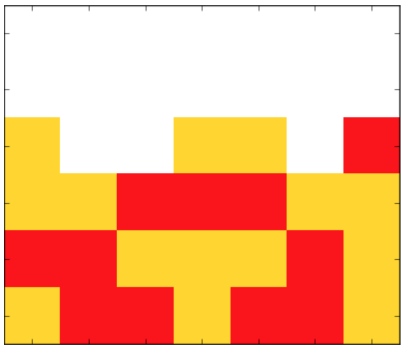

Figure 6. Visualisation of final representation of a processed board.

3.1.1 Object Detection

Figure 1 shows the visualisation of the object detection using Haar wavelet features. A more simple Local Binary Pattern (LBP) has been tried as well, but led to fewer successful detections.

3.1.2 Filters

Figure 2 shows the original image coming from the camera without any image processing except downscaling. Figure 3 and 4 show the binary channels of the board. Note that all processing steps following the object detection are carried out no matter whether the detection was successful or not.

3.1.3 Transformation

Figure 5 shows the perspective-transformed image with gridpoints superimposed. After this step, the final representation of the board can be seen in 6.

3.2. Learning

Our training algorithm was mainly CPU-based; using a GPU for tensor operation did not result in a speed-up since the time was not lost in the neural network computation but rather in the game logic, such as checks for winning position. We ported our Python game logic code to Cython code but were unable to achieve a speed-up without investing significantly more engineering effort. In the end, we were able to simulate $N_g = 250$ games and learn from them within 4s on a fast desktop CPU (i7-4790K @ 4.00 GHz) or within 8s on a regular notebook CPU (i5-3210M CPU @ 2.50GHz).

Figure 7 shows the evolution of the winning rate during 1000 training cycles. As one can easily see, the winning rate converges rather quickly to $70\tilde{}80\%$ in self-play. Literature suggests that there might be a strong improvement after a huge number of iteration cycles ($> 10^7$) but due to limited project time and server computation resources we haven't been able to research this further.

We tested different ways to propagate the reward used for the reinforcement learning algorithm. In most cases, we just assigned the total reward r_{total} to the last move of the game and did not explicitly specify a reward for all other moves. In one experiment, we instead set the reward individually for each move. As Figure 8 shows, this does not visibly change the results.

Quite unsurprisingly, the winning rate is a lot higher with around 90% if we play against a completely random opponent instead of our algorithm itself (see Figure 9). Even if we modify the random opponent in a way that makes it choose a winning move if possible in the next move, the winning rate still remains near 90% (see Figure 10).

However, even though 70% winning rate against self-play and 90% winning rate against a random player sounds

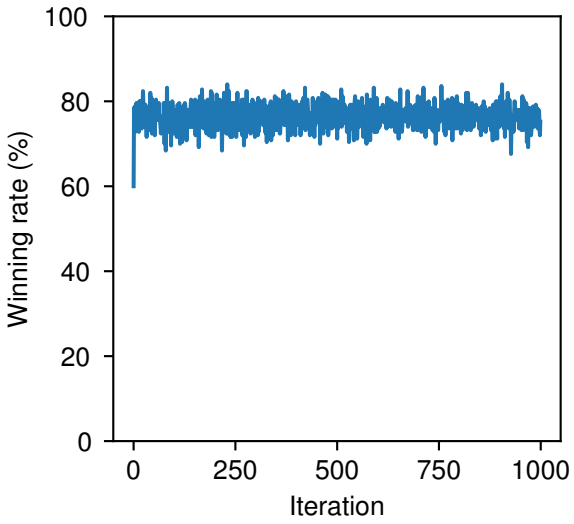

Figure 7. Training with $N_g = 250$, self play with 1% noise, reward given for last move.

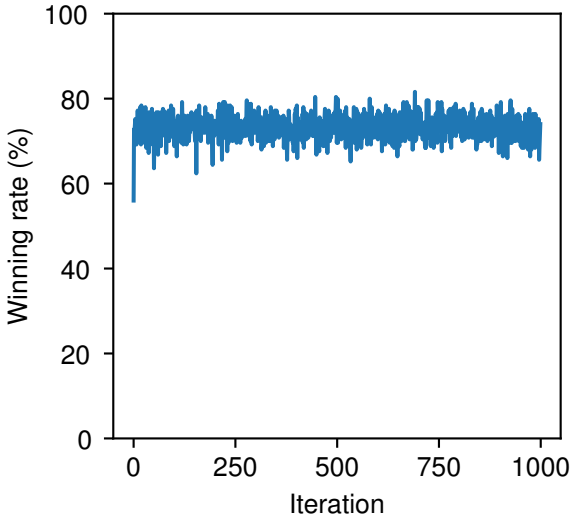

Figure 8. Training with $N_g = 250$, self play with 1% noise, reward given for all moves.

somewhat promising on first sight, playing against human players shows that the algorithm tends to employ mostly very simple strategies along the lines of building columns of coins in the middle and can rather easily be tricked by a human player.

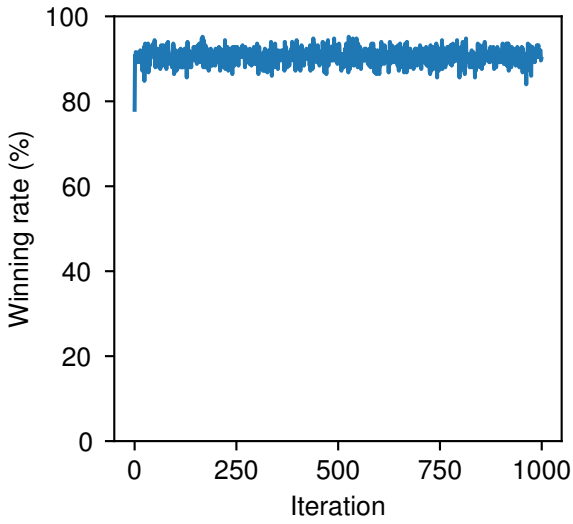

Figure 9. Training with $N_g = 250$, randomly playing opponent, reward given for last move.

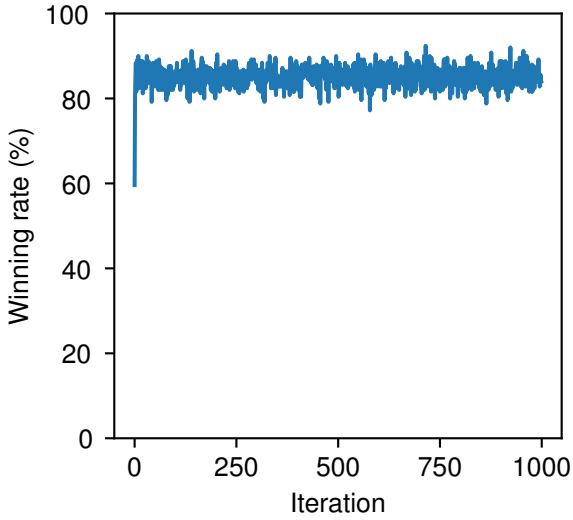

Figure 10. Training with $N_g = 250$, randomly playing opponent, reward given for last move.

4. Discussion

We have not been able to train our neural network to play the game well enough to reasonably compete with human players. This is probably due to a number of reasons, including the basic design of our neural network and the local minima our algorithm appears to be trapped in.

While Thill et. al. [8] and Ghory [9] show that it is possible to achieve good results with reinforcement learn-

ing on Connect Four, it requires more advanced methods as outlined in their papers and therefore significantly more engineering effort and computation time.

Building on our work, it would be interesting to replicate the results of the aforementioned authors as their code is not publicly available. It would also be interesting to train against a perfect or near-perfect opponent. Unfortunately, we haven't been able to find an implementation of the perfect algorithm that we could use for this purpose and it would have stretched the outlines of this project to create one.

Our work indicates that in principle, it is possible to solve Connect Four with reinforcement learning methods, but also shows that it would be more straightforward in terms of engineering and computation effort to create a classical AI based e.g. on minimax algorithms. This promises to also be able to beat most human players.

4.1. Vision

Apart from learning algorithms, image recognition was also employed for the purpose of this work. It should be stated that for a better generalization, modifications to the latter should be performed so as to guarantee a reliable board recognition even when it is placed in front of more challenging backgrounds. Machine learning based vision was employed in form of the Haar wavelet object recognition, whereas colour matching and object transformation were still performed in a classical way and may need to be adapted for different environments and configurations. Color references might perform well in a semi-classical code manner, where a colour reference of a coin would be found by machine learned object detection and then used for classical filters. The use of a smartphone would also allow for hardware and user feedback (i.e. turn on camera light, giving advise or show boxes when detection was successful).

Object Detection More samples and variations of them would certainly allow for a more robust solution. One possible way of achieving a more extensive dataset would be to use a webcam and record a video while turning around the board closely and in different environments.
For a proper end-user application, further sanity checks of the extracted board configuration should be implemented as well, possibly parallel to data-taking for the maximum user-friendliness.
When testing the algorithm on unseen images, we observe mixed results. Object recognition results in both too narrow (Figure 12) and too wide (Figure 11) detection boxes which can decrease overall recognition. However there is no reason known by the authors which would not allow the object recognition code for greater generalisation. Given even small amounts of additional training data in different environments would probably allow for better performance.

Figure 11. Unseen test image. Detection box too wide.

Figure 12. Unseen test image. Detection box too narrow.

References

[1] S. Edelkamp and P. Kissmann, "Symbolic classification of general two-player games," in *KI 2008: Advances in Artificial Intelligence: 31st Annual German Conference on AI, KI 2008, Kaiserslautern, Germany, September 23-26, 2008. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 185–192, ISBN: 978-3-540-85845-4. DOI: 10.1007/978-3-540-85845-4_23. [Online]. Available: http://www.tzi.de/~edelkamp/publications/conf/ki/EdelkampK08-1.pdf.

[2] V. Allis, "A knowledge-based approach to connect-four. the game is solved: White wins," in *Master's thesis, Vrije Universiteit*, 1988, p. 4.

[3] J. D. Allen, *Expert play in connect-four*, 1990. [Online]. Available: https://tromp.github.io/c4.html.

[4] *Opencv.org*. [Online]. Available: https://opencv.org/license.html.

[5] R. Lienhart, A. Kuranov, and V. Pisarevsky, "Empirical analysis of detection cascades of boosted classifiers for rapid object detection," *Proceedings of the 25th DAGM Pattern Recognition Symposium*, pp. 297–304, 2003, ISSN: 0021-9541. DOI: 10.1007/978-3-540-45243-0_39.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980.

[7] *Pytorch*. [Online]. Available: http://pytorch.org/.

[8] M. Thill, P. Koch, and W. Konen, "Reinforcement learning with n-tuples on the game connect-4," in *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature - Volume Part I*, ser. PPSN'12, Taormina, Italy: Springer-Verlag, 2012, pp. 184–194, ISBN: 978-3-642-32936-4. DOI: 10.1007/978-3-642-32937-1_19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32937-1_19.

[9] I. Ghory, *Reinforcement learning in board games.* 2004. [Online]. Available: https://www.cs.bris.ac.uk/Publications/Papers/2000100.pdf.